Taxiunternehmen

Aufgaben

Für ein Taxiunternehmen mit einer Flotte von 150 Fahrzeugen sollen Anwendungen für die Zentrale und die Taxameter sowie eine Datenbank zur Verwaltung der Taxifahrerinnen und Taxifahrer (im Folgenden Fahrer genannt), Fahrzeuge und Einsätze entwickelt werden.

1 Entwicklung von Software für die Taxizentrale und zur Steuerung der Taxameter.
Für die Auftragsverwaltung der Taxizentrale soll ein Softwaresystem entwickelt werden. Außerdem ist eine Steuerung für die in den Taxis eingebauten Taxameter erforderlich.

1.1 Die Entwicklung von Software ist ein strukturierter Prozess, der in mehreren Phasen erfolgt. Ziel ist die Sicherstellung der Softwarequalität.

1.1.1 Folgende Phasen der Softwareentwicklung lassen sich abgrenzen:
- Analysieren und Definieren der Anforderungen
- Softwareentwurf
- Implementierung und Komponententests
- Integration und Systemtests
- Betrieb und Wartung

Beschreiben Sie die Aufgaben, die in den einzelnen Phasen zu erledigen sind. Nennen Sie jeweils ein Produkt (z.B. Dokument oder Diagramm), das in der jeweiligen Phase zu erstellen ist.

(7 BE)

1.1.2 Beschreiben Sie vier Qualitätskriterien, um Softwarequalität sicherzustellen.

(4 BE)

1.2 Kundinnen und Kunden (im Folgenden Kunden genannt) können bei der Taxizentrale telefonisch rund um die Uhr Aufträge für Fahrten, unter Angabe der Start- und Zieladresse, aufgeben. Für die Auftragsabwicklung in der Taxizentrale liegt ein Anwendungsfalldiagramm vor (Material 1).
Beschreiben Sie die wesentlichen Notationselemente des vorliegenden UML-Anwendungsfalldiagramms sowie die Unterschiede zwischen extend- und include-Beziehungen. Erläutern Sie die inhaltliche Bedeutung des vorliegenden Anwendungsfalldiagramms.

(5 BE)

1.3 Die Auftragsdaten einer Fahrt werden vom Fahrer eingegeben bzw. vom Taxameter ermittelt und nach Abschluss des Auftrags auf dem Electronic Key des Fahrers gespeichert. Alle Auftragsdaten werden regelmäßig in der Taxizentrale ausgelesen und die entsprechenden Auftragsobjekte erzeugt. Ein erstes UML-Klassendiagramm für die Taxizentrale finden Sie in Material 2, die Dokumentation der Klassen `DateTime` und `List` in Material 3.

1.3.1 Bei der Erzeugung eines Auftrags werden das Taxi, der Fahrer sowie die Auftragsdaten in der Form:
`Sonnenweg 15,60487,Frankfurt a.M.; Markt 17,60311,Frankfurt a.M.;`
`14.04.2022 16:28:50;14.04.2022 17:10:50;20.0;42.25`
übergeben. Die durch Semikolon getrennten Elemente stehen für Startadresse, Zieladresse, Start- und Endzeit der Fahrt, gefahrene Kilometer und Fahrpreis.
Überführen Sie die Klasse `Auftrag` in eine objektorientierte Programmiersprache und implementieren Sie den Konstruktor.

(5 BE)

1.3.2 Die Methode `bearbeiteAuftrag()` der Klasse `TaxiZentrale` sucht nach freien Taxis, die sich zur angefragten Zeit am nächsten an der Startadresse (Parameter `von`) befinden. Für die ermittelten Taxis wird beim Fahrer angefragt, ob er bereit ist für die Annahme des Auftrags. Signalisiert der Fahrer seine Bereitschaft, wird der Auftrag an ihn vergeben. Konnte ein Taxi ermittelt werden, gibt die Methode `true` zurück, ansonsten `false`.
Entwickeln und zeichnen Sie ein Sequenzdiagramm für die Methode.

Hinweis: Eine Vorlage ist in Material 4 zu finden.

(8 BE)

1.3.3 Die Methode `getTaxisSortiertNachStandort(von: Adresse)` der Klasse `TaxiZentrale` gibt eine Liste mit bis zu fünf Taxis zurück, die sich am nächsten von der Adresse `von` befinden, wobei das naheste Taxi zuerst aufgeführt ist. Es sind nur die Taxis zu berücksichtigen, die zu der angefragten Zeit frei sind.
Implementieren Sie die Methode.

(10 BE)

1.4 In jedes Taxi ist ein Taxameter eingebaut. Seine Komponenten sind in Material 5 dargestellt. Das UML-Klassendiagramm für das Taxameter finden Sie in Material 6.

1.4.1 Die Methode `calculatePrice()` der Klasse `Taxameter` ermittelt die Fahrkosten anhand der zurückgelegten Kilometer, wobei folgende Entgelttabelle zu berücksichtigen ist:

Beförderungsentgelte	**Betrag in EUR**
Grundpreis	3,50
Wegstreckenentgelt je km für die ersten 15 km	2,00
Wegstreckenentgelt je km ab dem 16. km	1,75

Entwickeln und zeichnen Sie ein Struktogramm für den Algorithmus der Methode.

(5 BE)

1.4.2 Die Daten eines Auftrags werden als Byte-Array von der Methode `writeData()` der Klasse `EksAdapter` auf den Electronic Key geschrieben.
Zum Aufbau der Verbindung wird das Steuerzeichen STX (Start of Text) auf die serielle Schnittstelle geschrieben. Lautet die Antwort von der Schnittstelle DLE (Data Link Escape) werden die zu speichernden Daten in der folgenden Form geschrieben:

Datenblock							
1. Zeichen	**2. Zeichen**	**3. Zeichen**	**...**	**n. Zeichen**	**DLE**	**ETX**	**Checksum**
1 Byte	1 Byte	1 Byte		1 Byte	1 Byte	1 Byte	1 Byte

Lautet die darauf folgende Antwort DLE, wurde der Datenblock fehlerfrei übernommen. Ist die Antwort hingegen das Steuerzeichen NAK (Negative Acknowledgement), beginnt die Steuerung das Schreiben erneut mit dem Verbindungsaufbau STX. Bei der Antwort EM (End of Medium) steht kein ausreichender Speicherplatz zur Verfügung und der Vorgang wird abgebrochen. Die Methode gibt den Fehlerstatus wie folgt zurück:
0 ohne Fehler
1 Datenträger ist voll
2 Schnittstelle ist nicht geöffnet oder Datenträger fehlt
Implementieren Sie die Methode.

Hinweise: Das Steuerzeichen ETX steht für End of Text. Die Dokumentation der Klasse `Serial` ist in Material 3 zu finden.

(6 BE)

1.4.3 Die `run()`-Methode der Klasse `Taxameter` steuert die Auftragsabwicklung im Taxi. Der Fahrer bedient das Taxameter über entsprechende Tasten. Für den Ablauf der Methode ist folgendes zu berücksichtigen:
– Solange der EKS-Adapter keinen Electronic Key erkennt, befindet sich das Taxi im Status „Nicht in Betrieb" und die Fahrernummer hat den Wert -1.
– Wenn der EKS-Adapter einen Electronic Key erkannt hat, liest er die Fahrernummer ein.
– Solange der Electronic Key vorhanden ist, kann der Fahrer die Tasten bedienen, die das Taxameter regeln.
– Taste 1: Das Taxi hat den Status „frei". Das Dachzeichen ist angeschaltet.
– Taste 2: Der Fahrer hat einen Auftrag von der Taxizentrale erhalten. Das Taxi hat den Status „Anfahrt Kunde". Das Dachzeichen ist ausgeschaltet.
– Taste 3: Das Taxi hat einen Kunden aufgenommen und der Status ist „Anfahrt Ziel". Der aktuelle Kilometerstand, die Startadresse sowie Startzeit werden festgehalten. Das Dachzeichen ist ausgeschaltet.
– Taste 4: Das Ziel ist erreicht. Der aktuelle Kilometerstand, die Zieladresse sowie die Endezeit werden festgehalten. Der Fahrpreis wird berechnet und auf dem Display ausgegeben. Die Auftragsdaten werden in der Form `"[wagenNr];[fahrerNr];[von];[nach];[start];[ende];[fahrstrecke];[fahrpreis]"` auf dem Electronic Key des Fahrers gespeichert.
– Taste 5: Der Kunde wünscht eine Quittung.
Implementieren Sie die Methode.

Hinweis: Fehlerfälle durch das Drücken einer falschen Taste während der Durchführung eines Auftrags müssen nicht berücksichtigt werden.

(10 BE)

2 Datenbank für die Einsatzverwaltung und -planung des Taxiunternehmens
Ein erstes Entity-Relationship-Modell (ERM) ist in Material 7 dargestellt.

2.1 Überführen Sie das ERM in das relationale Modell in der 3. Normalform und begründen Sie Ihre Entscheidungen.

Hinweis: Alle Relationen sind in der Schreibweise Relation `(PK, Attribut, ... FK#)` anzugeben.

(6 BE)

2.2 Einige Daten der Datenbank sollen ergänzt, geändert bzw. ausgewertet werden.

Hinweise: Die Funktion `CONCAT(arg1, arg2,...)` gibt die verknüpfte Zeichenfolge der übergebenen Zeichenketten zurück.
Die Funktion `TIMESTAMPDIFF(MINUTE, datetime_expr1, datetime_expr2)` liefert das Ergebnis der Subtraktion `datetime_expr2 - datetime_expr1` in Minuten.

2.2.1 Das Unternehmen stellt Lennard Knebel als neuen Taxifahrer ein. Die Fahrernummer ist 333, sein Stundenlohn beträgt 12 Euro. Sein erster Einsatz ist für den 18.05.2023 von 8 bis 17 Uhr geplant. Das Taxi für diesen Einsatz hat das Kennzeichen F-TT 661.
Formulieren Sie die entsprechenden SQL-Anweisungen, um die Daten in die Datenbank einzufügen.

(5 BE)

2.2.2 Der Fahrer Tobias Tischendorf ist erkrankt und soll für seine Einsätze vom 01.05.2023 bis 14.05.2023 durch Lennard Knebel (Fahrernummer 333) ersetzt werden.
Formulieren Sie eine SQL-Anweisung zur Aktualisierung der Datenbank.
Hinweis: Fahrernamen sind immer eindeutig.

(3 BE)

2.2.3 Alle Taxis, die noch keinen Einsatz am 30.05.2023 haben, sollen ermittelt werden. Entwickeln Sie eine entsprechende SQL-Anweisung.

(3 BE)

2.2.4 Für den Fahrer Karl-Heinz Großfuss soll der Stundenzettel für den Monat März ausgegeben werden. Beispiel:

Arbeitszeit von - bis	**Anzahl Stunden**
2023-03-01 08:00:00 - 2023-03-01 17:00:00	9.00
2023-03-03 08:00:00 - 2023-03-03 16:00:00	8.00
2023-03-04 08:00:00 - 2023-03-04 17:15:00	9.25
2023-03-05 12:00:00 - 2023-03-05 22:00:00	10.00
2023-03-06 20:00:00 - 2023-03-07 06:00:00	10.00
...	

Entwickeln Sie eine entsprechende SQL-Anweisung.
Hinweis: Fahrernamen sind immer eindeutig.

(4 BE)

2.2.5 Implementieren Sie eine SQL-Anweisung, die eine nach Fahrernamen sortierte Liste mit IBAN und jeweiligem Monatslohn des betreffenden Fahrers für März 2023 ausgibt. Zu berücksichtigen sind die Einsätze, die im März 2023 enden.

(4 BE)

2.3 Die Datenbank soll um folgende Anforderungen erweitert werden:

- Ein Taxi hat zusätzlich die Eigenschaften Erstzulassung und Kilometerstand.
- Ein Taxi gehört zu einem Fahrzeugmodell, welches mit der Modellbezeichnung, dem Hersteller und der Anzahl der Sitzplätze zu speichern ist. Die Antriebsart eines Modells kann elektrisch oder durch Verbrennung eines Kraftstoffs (Benzin, Diesel oder Gas) sein. Ein Hybridfahrzeug hat eine Kombination von mehreren Antriebsarten (z.B. elektrisch und Diesel).
- Taxis können Schäden aufweisen, die mit dem Aufnahmedatum, der Schadenstelle (z.B. Kotflügel vorne links) und der Schadensart (z.B. Kratzer) festzuhalten sind.
- Eine von einer Werkstatt durchgeführte Reparatur eines Schadens wird mit Datum und Kosten gespeichert. Eine Werkstatt hat einen Namen und eine Adresse sowie mindestens eine Ansprechperson mit Namen und Telefonnummer.

Entwickeln und zeichnen Sie ein ERM, das die genannten Anforderungen umsetzt.

Hinweise: Das ERM ist mit Entitätstypen, Attributen und Beziehungen sowie deren Kardinalitäten in [min, max]-Notation darzustellen.

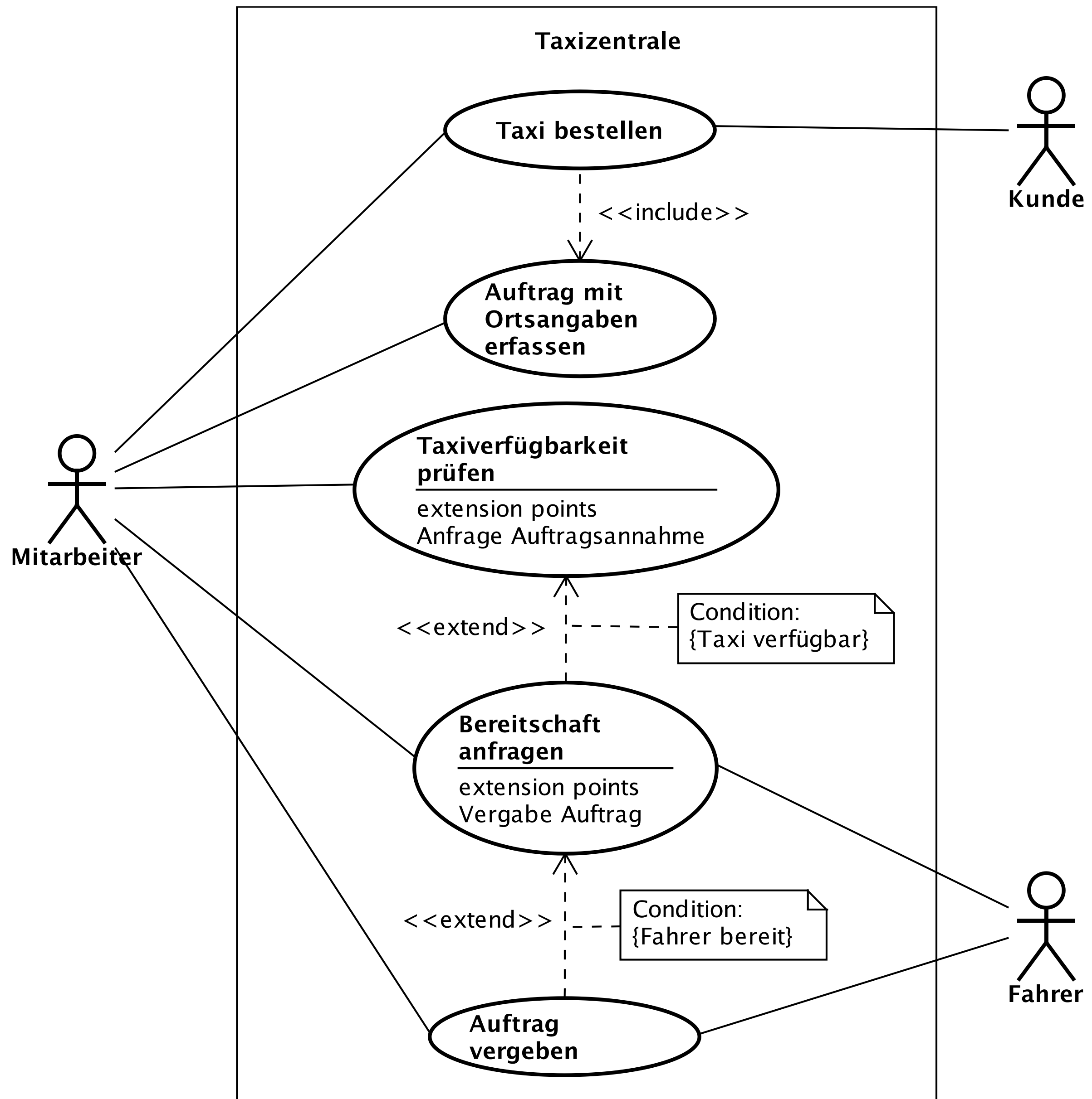
(10 BE)

2.4 Die Tabelle in Material 8 enthält Redundanzen und Inkonsistenzen. Nennen Sie diese und beschreiben Sie mithilfe von Beispielen aus der Tabelle die verschiedenen Arten von Anomalien.

(5 BE)

Material 1

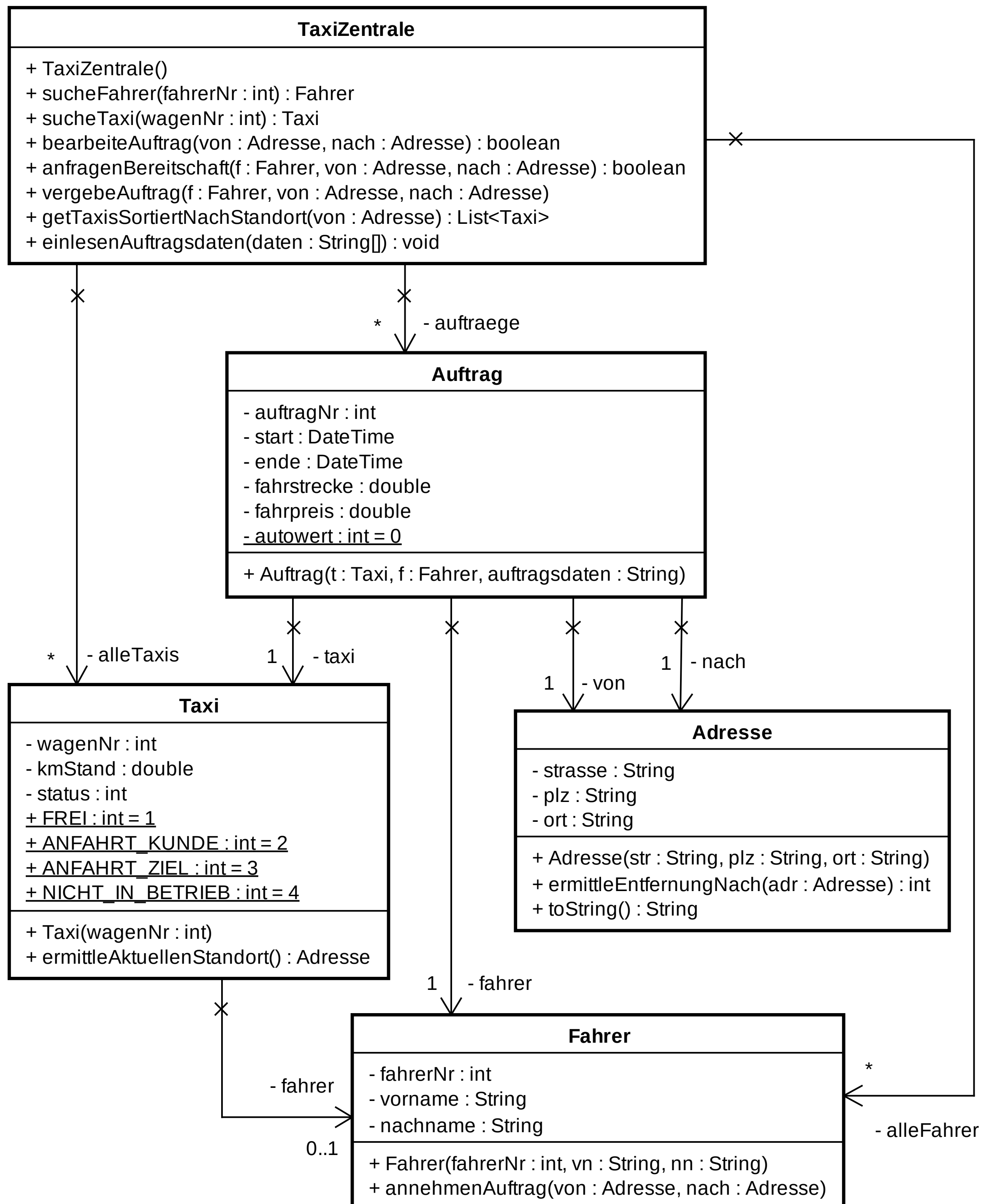
UML-Anwendungsfalldiagramm Taxizentrale



Hinweis: Mitarbeiterinnen und Mitarbeiter werden im Folgenden Mitarbeiter genannt.

Material 2

UML-Klassendiagramm Taxizentrale



Hinweise: Der Status eines Taxis kann frei, Anfahrt zum Kunden bzw. zum Ziel des Auftrags oder nicht in Betrieb sein. Auf alle Attribute kann mittels get-Methoden zugegriffen werden.
Die Methoden `ermittleEntfernungNach(adr:Adresse)` der Klasse `Adresse` gibt die Entfernung zwischen den Adressen in Metern zurück.
Die Methode `toString()` der Klasse `Adresse` liefert die String-Repräsentanz des `Adresse`-Objekts in der Form "`[strasse],[plz],[ort]`".

Material 3

Klassendokumentationen

Klasse `DateTime`

DateTime
+ DateTime() + DateTime(datetime : String) + toString() : String

`DateTime()`
erzeugt ein `DateTime`-Objekt mit dem aktuellen Systemdatum und der Systemuhrzeit im Format `dd.mm.yyyy hh:mm`.

`DateTime(datetime: String)`
erzeugt ein `DateTime`-Objekt mit den Werten aus `datetime` im Format `dd.mm.yyyy hh:mm`.

`toString(): String`
liefert die String-Repräsentanz des `DateTime`-Objekts im Format "`dd.mm.yyyy hh:mm`".

Klasse `List`

List<T>
+ List<T>() + add(obj : T) + add(index : int, obj : T) + get(index : int) : T + remove(index : int) : T + size() : int

`List<T>()`
erzeugt eine generische Liste mit Elementen des Typs `T`.

`add(obj: T)`
hängt das Objekt `obj` vom Typ `T` am Ende der Liste an.

`add(index: int, obj: T)`
fügt das Objekt `obj` vom Typ `T` an der Position `index` in die Liste ein.

`get(index: int): T`
liefert das Listenelement an der Position `index` zurück bzw. `null`, falls `index` negativ oder größer gleich der Anzahl der momentan enthaltenen Elemente ist.

`remove(index: int): T`
entfernt das Objekt vom Typ `T` an der Position `index` aus der Liste und gibt es zurück.

`size(): int`
liefert die Anzahl der Elemente in der Liste zurück.

Klasse `String`

String
+ getBytes() : byte[ ] + length() : int + split(str : String) : String[ ]

`getBytes(): byte[]`
wandelt den String in ein `byte`-Array um und gibt es zurück.

`length(): int`
liefert die Anzahl der Zeichen des Strings.

`split(str: String): String[]`
teilt einen String am Trennzeichen `str`. Die Teil-Strings werden in einem Feld zurückgeliefert.

Die Klasse `Serial`

Serial
- portName: String - baudrate: int - dataBits: int - stopBits: int - parity: int
+ Serial(String portName, int baudrate, int dataBits, int stopBits, int parity) + open(): boolean + close() + dataAvailable(): int + read(): byte + read(b: byte[], len: int): int + readLine(): String + write(value: int) + write(b: byte[], len int) + write(s: String)

Ein Exemplar der Klasse `Serial` ermöglicht die Kommunikation über die serielle Schnittstelle

`Serial(…)`
der Konstruktor initialisiert die serielle Schnittstelle ohne sie zu öffnen.

`portName:`	Name des Ports
`baudrate:`	Baudrate
`dataBits:`	Datenbitanzahl
`stopBits:`	Stopbitanzahl
`parity:`	Parität

`open(): boolean`
öffnet die serielle Schnittstelle; liefert `true`, wenn die Schnittstelle verwendbar ist.

`close()`
schließt die serielle Schnittstelle; die Schnittstelle ist dann solange nicht mehr verwendbar, bis sie wieder geöffnet wird.

`dataAvailable(): int`
liefert die Anzahl der Bytes, die von der seriellen Schnittstelle gelesen werden können.

`read(): byte`
liest ein Byte (0..255) von der serielle Schnittstelle, bzw. –1, wenn die Schnittstelle nicht geöffnet ist. Die Methode blockiert, bis ein Byte verfügbar ist.

`read(b: byte[], len: int)`
liest mehrere Bytes von der seriellen Schnittstelle in ein Byte-Array; liefert die Anzahl der gelesenen Bytes, bzw. –1, wenn der Schnittstelle nicht geöffnet ist. Die Methode blockiert, bis mindestens ein Byte verfügbar ist.

`readLine(): String`
liest eine Zeile von der serielle Schnittstelle, bzw. liefert null, wenn die Schnittstelle nicht geöffnet ist. Eine Zeile wird durch ein Zeilenendezeichen abgeschlossen; das Zeilenendezeichen wird jedoch nicht in den zurückgegebenen String übernommen. Die Methode blockiert, bis eine komplette Zeile eingelesen ist.

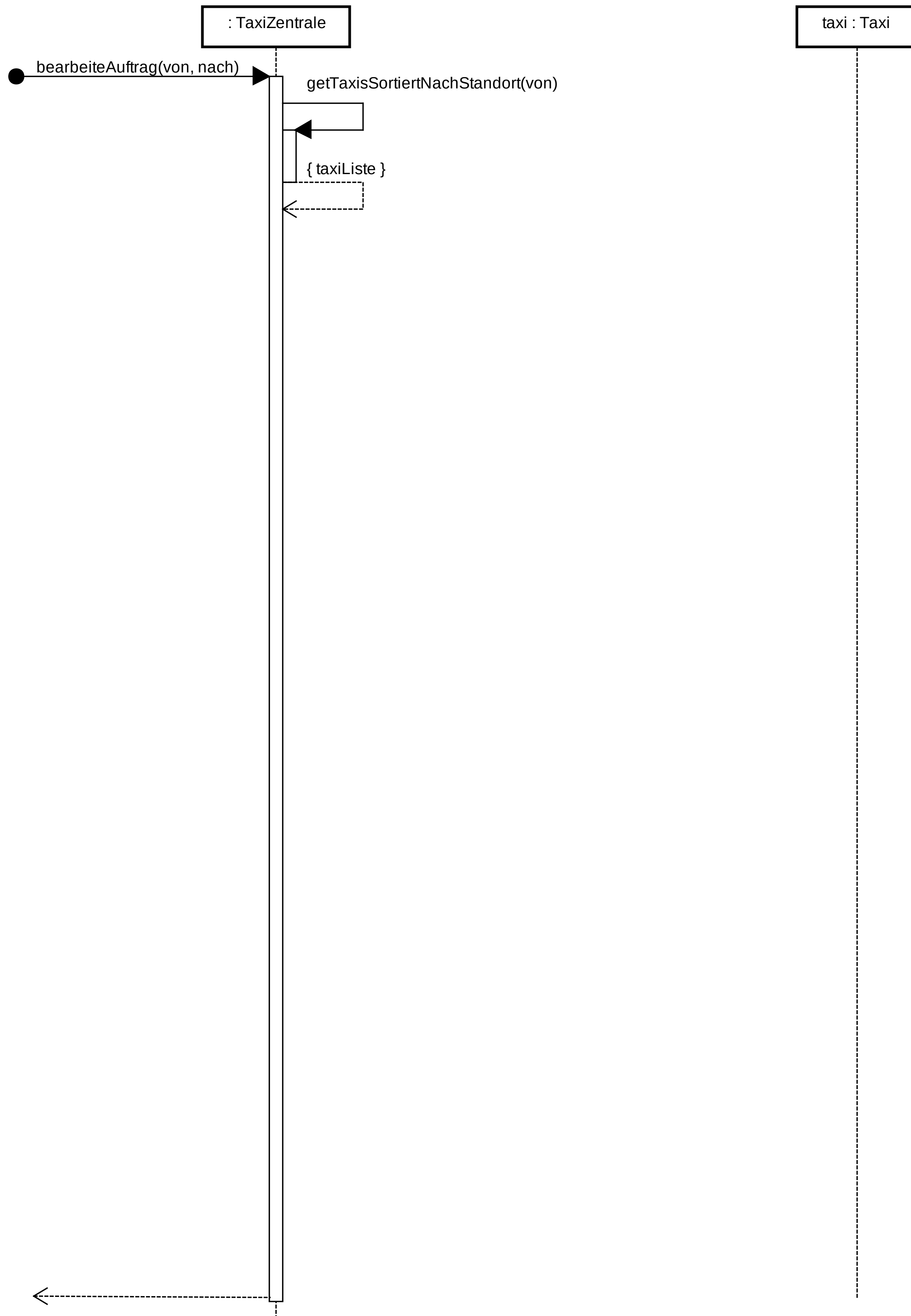
`write(value: int)`
schreibt ein Zeichen auf die serielle Schnittstelle; ist die Schnittstelle nicht geöffnet geschieht nichts.

`write(b: byte[], len: int)`
schreibt mehrere Bytes auf die serielle Schnittstelle; ist die Schnittstelle nicht bereit geschieht nichts.

`write(s: String)`
schreibt einen String auf die serielle Schnittstelle; ist die Schnittstelle nicht geöffnet geschieht nichts.

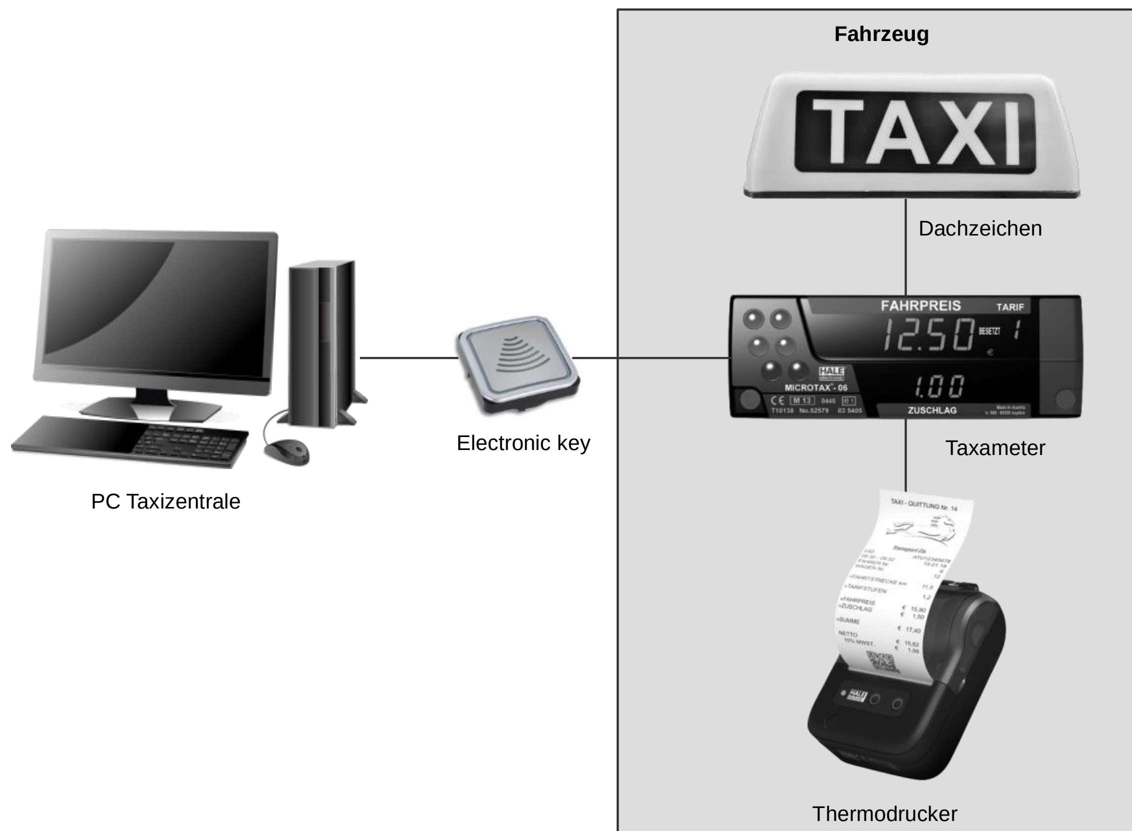
Material 4

Vorlage UML-Sequenzdiagramm



Material 5

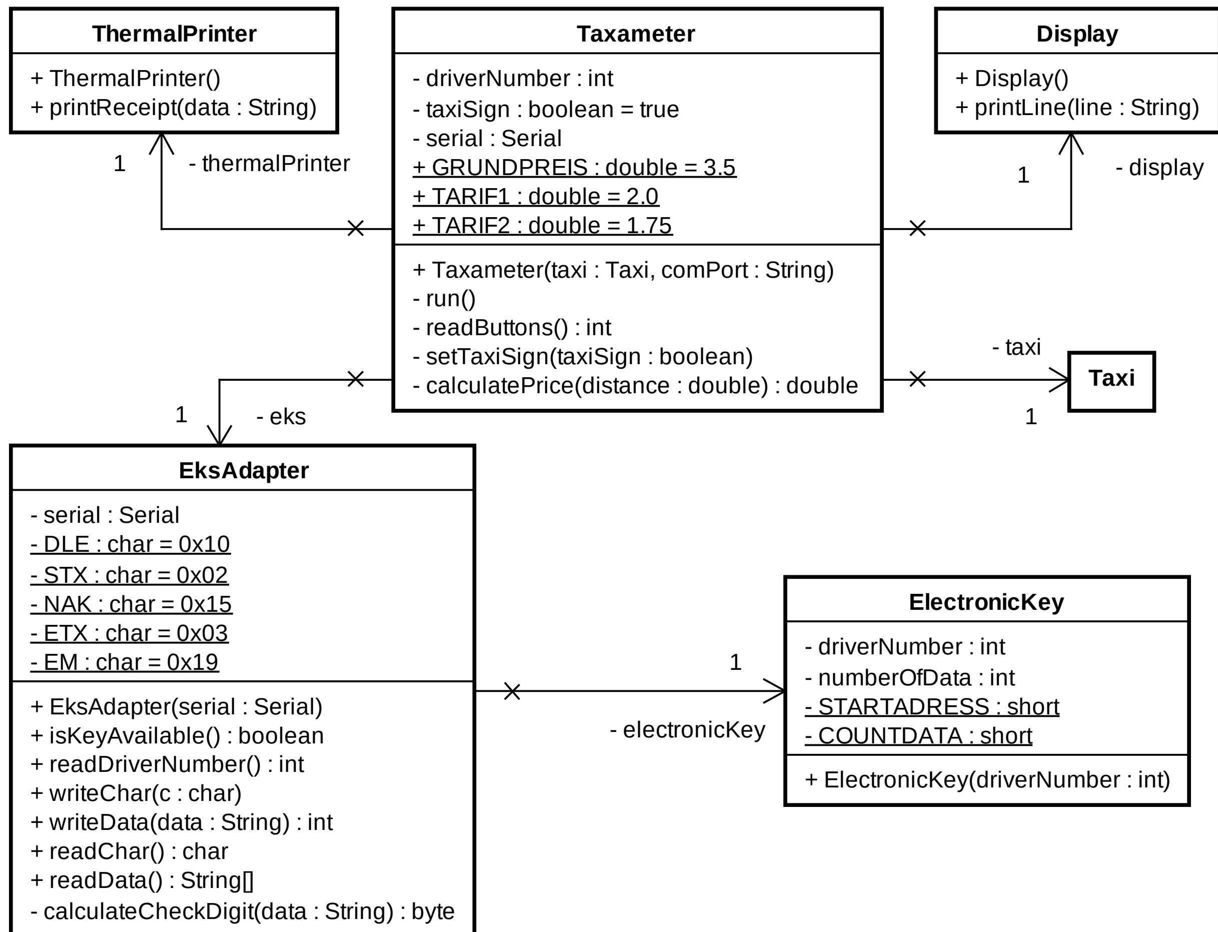
Beschreibung Taxameter und Komponenten



Hinweise: Ein Taxameter ist ein elektronisches Gerät zur Erfassung von Fahrpreisen auf Basis von Taxitarifen. Angeschlossen sind das Dachzeichen und ein Thermodrucker zur Quittungserstellung.
Die Anmeldung des Fahrers im Fahrzeug erfolgt durch ein Electronic-Key-System (EKS). Dabei handelt es sich um ein transponderbasiertes Schreib- und Lesesystem, also eine Kombination aus Schlüssel und Datenspeicher. Das EKS besteht aus dem Schlüssel (Electronic Key) und dem im Taxameter verbauten EKS-Adapter zur Schlüsselaufnahme.
Die Kommunikation zwischen Taxizentrale und Fahrern geschieht über Funk und ist nicht Gegenstand dieser Aufgabe.

Material 6

UML-Klassendiagramm Taxameter



Hinweise zum Klassendiagramm:

Klasse `Taxameter`
- `run()` steuert die Auftragsabwicklung.
- `readButtons()` wartet auf einen Tastendruck und liefert die gedrückte Taste am Taxameter.
- `setTaxiSign()` schaltet das Dachzeichen ein (`true`) oder aus (`false`).
- `calculatePrice()` liefert die Fahrkosten anhand der zurückgelegten Kilometer.

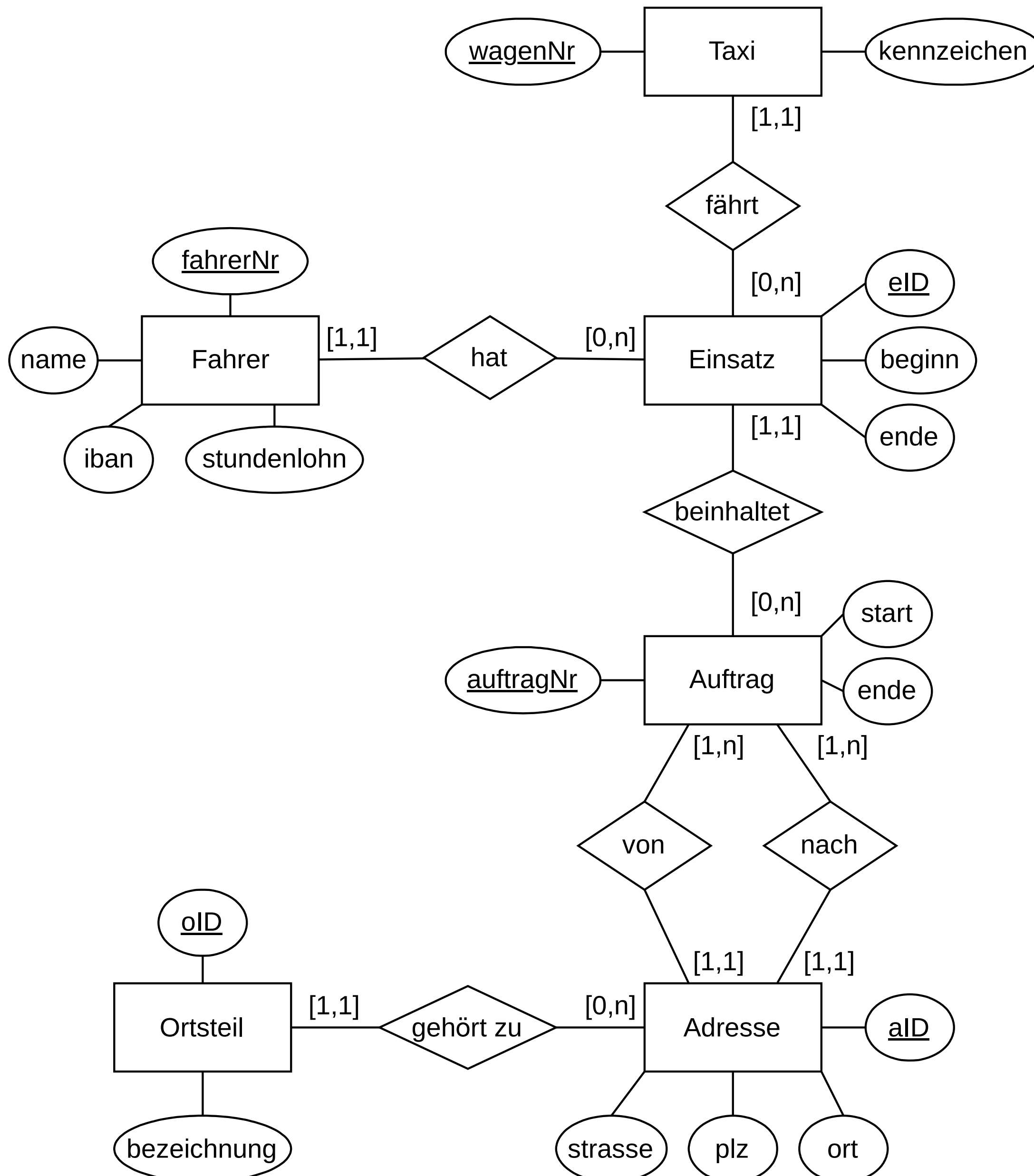
Klasse `EksAdapter`
- `isKeyAvailable()` prüft, ob ein Electronic Key erkannt wurde.
- `readDriverNumber()` liefert die auf dem Schlüssel gespeicherte Fahrernummer.
- `writeChar()` schreibt ein Zeichen auf die serielle Schnittstelle.
- `writeData()` speichert die übergebenen Auftragsdaten auf dem Electronic Key.
- `readChar()` liest ein Zeichen von der seriellen Schnittstelle.
- `readData()` liefert ein String-Array mit den auf dem Electronic Key gespeicherten Auftragsdaten.
- `calculateCheckDigit()` ermittelt die Checksumme für den übergebenen String.

Klasse `Display`
- `printLine()` dient der Textausgabe auf dem Display des Taxameters.

Klasse `ThermalPrinter`
- `printReceipt()` erstellt die Quittung aus den übergebenen Auftragsdaten und druckt sie aus.

Material 7

ERM Einsatzplanung



Hinweise: Die Attribute `beginn`, `ende` und `start` werden im Format `YYYY-MM-DD HH:MM:SS` gespeichert. Die IDs (`eID`, `aID`, `oID`) werden automatisch vergeben (`AUTO_INCREMENT`).

Material 8

Tabelle Einsatzfahrten in Wiesbaden für 2023 (Ausschnitt)

Datum	Fahrer	Telefon Fahrer	Kennzeichen	Fahrzeug Typ	Startadresse	Zieladresse
12.04	Max Müller	0151 123456	WI-TF25	VW Caddy	Anglergasse 15	Mainzer Str. 174
	Ina Ach	0160 667788	WI-TF28	Toyota Corolla	Rheinblickstraße 26	Wupperstraße 6
13.04	Max Müller	0151 1234	WI-TF25	VW Caddy	Mümmelmannweg 2	Buchenstraße 2
	Ina Ach	0160 667788	WI-TF28	Toyota Corolla	Kirchgasse 28	Am Kupferberg 1
14.04	Finn Kager	0160 876541	WI-TF42	Mercedes Vito	Metzer Str. 8	Etzelstraße 22